An Extensible, Modular Architecture for Simulating Urban Development, Transportation, and Environmental Impacts

Michael Noth* Alan Borning

Dept. of Computer Science & Engineering, University of Washington, Box 352350, Seattle, Washington 98195, {noth,borning}@cs.washington.edu

Paul Waddell

Evans School of Public Affairs, University of Washington, Box 353055, Seattle, Washington 98195, pwaddell@u.washington.edu

* Corresponding author

Abstract

UrbanSim simulates the development of urban areas, including land use, transportation, and environmental impacts, over periods of twenty or more years. Its purpose is to aid urban planners, residents, and elected officials in evaluating the long-term results of alternate plans, particularly as they relate to such issues as housing, business and economic development, sprawl, open space, traffic congestion, and resource consumption. From a software perspective, it is a large, complex, system, with heavy demands for excellent space efficiency and support for software evolution. It consists of a collection of models that represent different urban actors and processes, an object store that holds the state of the simulated urban environment, a model coordinator that schedules models to run and notifies them when data of interest has changed, and a translation and aggregation layer that performs a range of data conversions to mediate between the object store and the models. The paper concludes with a discussion of the lessons learned regarding implicit invocation, object storage, and automatic code generation that yield acceptable space and time efficiency, as well as support for software evolution, within this architectural framework.

1 Introduction

Patterns of land use and available transportation systems play a critical role in determining the economic vitality, livability, and sustainability of urban areas. Transportation interacts strongly with land use. For example, automobile-oriented development may induce demand for more roads and parking (which in turn induces more automobile-oriented development), while compact urban environments may induce more walking and demand for transit. Both land use and transportation have significant environmental effects, in particular on emissions, resource consumption, and conversion of rural to suburban or urban land.

Good technical support can play an important role in fostering informed civic deliberation and debate on these issues. To aid urban planners, residents, and elected officials in evaluating alternate scenarios—packages of policies and investments—we want to simulate the effects of these scenarios on patterns of urban growth and redevelopment, of transportation usage, and resource consumption, over periods of twenty or more years.

Early attempts at comprehensive urban simulations in the 1960s and early 1970s were largely unsuccessful (Lee, 1973, 1994). Much has changed since then, both on the supply side (including dramatically improved hardware, theoretical advances in urban economics and other disciplines, and the emergence of a commercial GIS market serving urban and regional planners), and on the demand side (including public concern over sprawl, legal challenges to transportation plans made without considering their land use implications (Garret and Wachs, 1996), and regulatory requirements such as the Clean Air Act Amendments of 1990). As a result, there has been somewhat of a renaissance in interest in urban simulation modeling over the past decade.

However, in terms of planning agency practice, land use planning is still often poorly integrated with transportation planning, despite their strong interactions. While transportation models have been in routine use by metropolitan planning organizations for decades, the state of common practice in land use modeling, and in integrated land use and transportation modeling, is much less advanced than that for transportation modeling alone. (See Section 4 for a discussion of prior and related work.)

The UrbanSim system has been designed and implemented in response to these needs. It is a system for simulating the development of urban areas, including land use, transportation, and environmental impacts, over periods of twenty or more years (Waddell, 1998b; UrbanSim Group, 2000). From a software perspective, it is a large, complex application, with heavy demands for excellent space efficiency and support for software evolution. The system is fully operational and freely available via our web site at www.urbansim.org. It consists of around 130,000 lines of Java code for the core UrbanSim system; including the visualization, data preparation, and calibration tools, the total is approximately 200,000 lines, plus another 100,000 lines of automatically generated code. It has been applied to Eugene-Springfield, Oregon; Salt Lake City, Utah; and Honolulu, Hawaii, working with the planning organizations in those metropolitan regions. Application to other regions is underway. We have also done a historical validation of the system, with very good results, starting UrbanSim with 1980 data for Eugene-Springfield, running it through 1994, and comparing the results with what actually transpired (Waddell, 2000).

2 Overview of the UrbanSim Architecture

To simulate an urban region, UrbanSim employs a collection of interacting *models*, representing different actors and processes in the urban environment, such as residents, businesses, land developers, and transportation networks. Each model encodes the behavior of agents in the simulation, as well as the objects they operate upon, such as land parcels and buildings. Objects correlate directly with easily-identifiable objects in the real world, making it easier to reason about their properties and behaviors. Entities can be shared across models, as can the objects they operate upon. Much more than other urban modeling systems, the UrbanSim model is very disaggregate and has high data requirements. These requirements enable modeling of processes to be done at a fine level, which allows leveraging spatial data in a manner not possible with more aggregate systems. At the same time, this makes the design and implementation of the system more difficult from a software perspective. Figure 5 illustrates the architecture of the UrbanSim system.

In addition to the models, the other principal components of UrbanSim are a *model coordinator* that schedules models to run and notifies them when data of interest has changed, an *object store* that holds the shared representations of agents and other entities in the simulated world, and a *translation and aggregation layer* that performs a range of data conversions to mediate between the Object Store and the models. The models do not communicate directly with each other; rather, they communicate via shared data held in the Object Store, mediated by the translation and aggregation layer. This extensible, modular architecture supports system evolution, in particular replacing a model with a revised one, and creating and integrating new models. It allows models to define and share common sets of objects that they all operate upon, via the Object Store (regardless of the original source of the data), and also allows them to monitor changes to data fields, providing a convenient method for models to synchronize their actions.

A primary goal of this architecture is to move as much of the software complexity out of the individual models and into the supporting infrastructure as possible. This supporting infrastructure need be written just once, and can have the attention of an expert programmer. The models, on the other hand, are both numerous and frequently changing. Often, specifying them is difficult, requiring considerable domain-specific expertise, specialized data, and testing; the more one can relieve the model designers of programming burdens the better, so that they can concentrate on issues arising from the domain.

2.1 Models

Models represent different actors or processes in the urban environment. In addition to encapsulating the behavior of the actor or process, each model is also responsible for defining the set of object types it operates on, and the fields of those objects with which it is concerned. A model can specify that it wishes to share fields also declared by other models, thus providing one technique for data-level coupling and integration of models via the Object Store. A model can also declare new object types that encapsulate domain-specific data not previously declared (e.g., a water quality model might declare a nutrient load value). A model may specify a set of object types and fields it wishes to monitor for updates, creations, or deletions. Each model is also responsible for indicating how frequently it wishes to be executed; there are no external constraints on how frequently or regularly a model need run.

The design of the models is informed by research in urban economics, sociology, civil engineering, and other disciplines. A discussion of the theoretical basis for the various models is given in reference (UrbanSim Group, 2000).

2.1.1 Currently Implemented Models

A list of the models in the current version of UrbanSim is given below. Each model runs once per simulation year, unless otherwise noted. All of these models consist of a collection of domain-specific case-based rules or decision rules that are encoded in Java code. However, given the encapsulation provided by the system architecture, models could be represented in other ways as well.

Demographic Transition Model The Demographic Transition Model is responsible for modeling births and deaths in the simulated population of households. Externally imposed population control totals—based on predictions by demographers for the given region—are used to determine overall target population values, and can be specified in more detail by distribution of income groups, age, size, and presence or absence of children. This enables the modeling of a shifting population distribution over time. Iterative proportional fitting (Beckman et al., 1995) is used to determine how many households of each type are to be created or deleted. Newly created households are placed in limbo, to be placed in buildings later by the Household Location Choice Model. Households to be deleted to meet the control totals are selected at random, drawn preferentially from households in limbo.

Household Mobility Model The Household Mobility Model simulates the decision processes of households deciding whether to move. Movement probabilities are based on historical data. Once a household has decided to vacate, it is placed in limbo to indicate it has no current location, and the space it formerly occupied is made available. (The displaced household will be subsequently placed by the Household Location Choice Model if housing is available.)

Household Location Choice Model The Household Location Choice Model is responsible for determining a location for each household that has no current location (i.e., is in limbo). For each such household, a sample of locations with empty housing units is randomly selected from the set of all possible alternatives. Each alternative in the sample is evaluated for its desirability to the household, through a nested logit model encoding decision rules and metrics. The coefficients for each rule are calibrated to observed and historical data. The household is placed into its most desired location among those available.

Economic Transition Model The Economic Transition Model is responsible for modeling job creation and loss. Employment control totals are used to determine target employment values, and can be specified by distribution of business sector. As with the Demographic Transition Model, iterative proportional fitting is used to determine how many businesses of each type are created or lost. New jobs are placed in limbo, to be placed later by the Employment Location Choice Model. In the case of losses, jobs are selected at random to be removed, drawn preferentially from any job without a location (i.e. in limbo).

Employment Mobility Model The Employment Mobility Model determines which jobs will move from their current locations during a particular year. (Choices regarding job creation and loss, mobility, and location are of course made by businesses large and small. However, in the economic transition and employment models, we use individual jobs as the unit of analysis.) Movement probabilities are based on historical data. Once a job has been determined to vacate a building, the job is placed in limbo to indicate it has no current location, and the space it formerly occupied is made available. The displaced job will be subsequently placed by the Employment Location Choice Model.

Employment Location Choice Model The Employment Location Choice Model is responsible for determining a location for each job that has no location (i.e., is in limbo). For each such job, a sample of locations with empty square feet, or space in housing units for home-based jobs, is randomly selected from the set of all possible alternatives. Each alternative in the sample is evaluated for its desirability for the particular job, through a multinomial logit model encoding decision rules and metrics. The job is placed into its most desirable location, among those available.

Accessibility Model The Accessibility Model encapsulates the interface to a (possibly external) travel model. It is responsible for maintaining accessibility values for objects within each traffic analysis zone, including accessibility by residents and employees to shopping and other amenities, to employment, and to the central business district. (These accessibility values encode how long it takes to get somewhere via driving, transit, walking, etc.) The link between land use and the travel model is two-way, since different accessibility values from the travel model will influence the decisions of developers, employers, and residents, giving rise to different travel demands, which then feed back into the travel model. The external travel model provides travel times and utilities to the Accessibility Model. This external model is typically run only once every 5 simulated years or when there is a major change to the transportation system, since running it is relatively expensive and since its outputs generally change more slowly than other values in the simulation. However, the Accessibility Model itself is run yearly, allowing the activity levels to be updated annually.

Land Developer Model The Land Developer Model simulates the action of a developer making decisions about where and what kind of construction to undertake (if any), including both new development and redevelopment of existing structures. Each time it is run, the model iterates over all gridcells on which development is allowed. For each such gridcell, a list of possible transition alternatives is created (representing different development types), including the alternative of not developing. The probability for each alternative being chosen is calculated in a multinomial logit model, and one of the alternatives is selected through a Monte Carlo sampling. The logit model uses numerous indicators to calculate the logit probabilities, including local and regional vacancy rates, distance to highways, travel time to the central business district, and neighborhood economic and environmental characteristics. Upon choosing to develop or redevelop a particular gridcell, the model constructs a Development Event and asks the Model Coordinator to add it to the event queue.

Land Price Model The Land Price Model simulates the evolution of land prices at each grid cell as the characteristics of locations change over time. It is based on urban economic theory, which states that the value of location is absorbed into the price of land. The model is calibrated by measuring the effect of site, neighborhood, accessibility, and policy effects on land prices from historical data. It also measures the effects of short-term fluctuations in local and regional vacancy rates on land prices. Land prices influence the density and profitability of the various potential forms of real estate development, and their costs to buyers, so this component plays a central role in the interactions among the other models.

2.1.2 Temporal Scale Issues and Simplifications

UrbanSim provides a much more disaggregate and detailed simulation than other urban land use models. Even so, to keep the computation manageable, the model makes many simplifying assumptions. For example, the Demographic Transition Model, like most of the models, runs once per simulated year. Each simulated year, it adjusts the total population values and distributions, but in reality people are born and die, and move into and out of the region, every day. Similarly, the Land Price Model simulates the operation of the real estate market at a temporally aggregate level, adjusting prices once per year rather than continuously. (See (UrbanSim Group, 2000) for additional discussion of the theoretical basis for these design decisions and their consequences.)

Another kind of simplification is in regard to the modeling of behavior. For example, the current Household Mobility and Household Location Choice model the possibility of households becoming homeless in only a very simplified way, in that if the available housing stock runs out before all households are placed, those remaining households will be homeless.

Modeling these two processes in a more realistic way would have quite different implications for the software architecture. For example, implementing an auction-style, continuous-time simulation of the real estate market would put dramatic stresses on the architecture, requiring much more data and processing. On the other hand, modeling homelessness in a more realistic way introduces some difficult issues from a modeling perspective, since there are other causes of homelessness besides being priced out of the housing market, but accommodating this within the overall system architecture should present no particular difficulties.

2.1.3 Defining a Model

The description of a model consists of a Model Definition File, and separately, a Java class definition for the model. The Model Definition File includes the model's name, and the set of objects and object fields it reads and writes, along with flags indicating if the fields are to be shared with other models (i.e., sharing a field with an already-created model). Shared variables are specified explicitly in the model definition rather than implicitly through duplicated names in order to ensure that any duplicate use of an object field is deliberate.

This avoids a potential problem where commonly-used names may be used in independent models but with different semantics, and sharing the variables in that case would cause erroneous results. For example, one model might refer to "population" as a count of persons, and another as a count of households, in which case trying to share the field would produce inconsistencies. Information from all the relevant model definitions is combined to produce the definitions of the objects in the Object Store (Section 2.3.2).

The remainder of the model's functionality is specified by a Java class, which must be a subclass of the abstract class Model. The following are the key methods defined by Model, and which are overridden in concrete subclasses.

init Perform any model-specific initialization, including notifying the Model Coordinator which objects and fields it wishes to monitor for changes.

run Perform the work of running the model at the current simulated time.

nextScheduledRunTime Return a **float** indicating the next simulated time that the model wishes to be run.

onCreate Perform any needed bookkeeping if an object of interest has been created. This method is invoked if one of the objects in which this model has registered interest has been added to the Object Store.

onChange An object type or field that this model monitors has been changed—react accordingly.

onRemove Perform any needed bookkeeping if an object of interest has been removed from the Object Store.

If the degree of interaction between a new model and existing models can be expressed at the data level and there is a well-defined order between them (e.g. one model's outputs are always used as inputs by another model), then no additional information is required. For example, the output of the Demographic Transition Model (i.e. newly-created households that reside in limbo) is an input to the Household Location Choice Model, and this interaction is wholly defined at the data level (i.e. the existence of new households in limbo). If models need to be more tightly coupled, or operate on differing temporal scales, the data notification interface can be used. For example, a continuous-time model can be set to monitor changes to data fields it uses as inputs, and compute an updated set of outputs only when its inputs have changed. The Translation/Aggregation Layer can help with models that operate on different spatial scales, for example by aggregating from the parcel or grid-cell level to the zonal level. The key methods used in providing this functionality are the onCreate, onChange, and onRemove methods of Model (as defined above), and the postQuery and postUpdate methods of the Model Coordinator (Section 2.4), which pass information through the Translation/Aggregation Layer and on to the Object Store.

In addition to providing a mechanism for coupling several models using implicit invocation, another application of the model notification mechanism is to support the caching of frequently-accessed data within a model, rather than repeatedly accessing it from the Object Store. This can be helpful if a model needs to perform costly processing on a large amount of data, as it can compute the results once and recompute only what is needed as parts of the underlying data are changed. Data modification messages serve as notification that the model's cache is no longer valid, and supply the specific data element(s) which have changed.

For example, the Land Price Model maintains regional and zonal-level vacancy rates. These more aggregate vacancy rates change slowly over time, so the Land Price Model computes them once, and modifies them as needed as households and employees move about the region, rather than recomputing the aggregate vacancy rates every time the model runs. As the vast majority of employees and households remain where they are at each simulation step, this approach substantially reduces the overall number and size of queries to the Object Store.

2.2 Model Coordinator

The Model Coordinator is responsible for managing the collection of models present in a simulation. It is responsible for determining the execution order of models, resolving any data dependencies one model may have on another, and notifying a model when another model has changed data it is monitoring.

Some key methods defined by the Model Coordinator class are:

runSimulation Run the simulation once the event queue has been populated.
executeEvent Execute a single event (provided as an argument to this method).
getOrdering Determine a total ordering among a collection of events (provided as an argument to this method).

2.2.1 The Event Queue

The Model Coordinator maintains an event queue containing timestamped events. These events include requests by a model to execute at a future time, development events scheduled to occur at a future time, database updates that were created by exogenous events that did not occur instantaneously, and policy events that indicate planned changes in regional or local policy. Running the simulation consists of gathering the set of events that are to occur at the current timestep, determining a total order for those events that preserves any data or ordering dependencies they may have, and then executing them in that order.

The event queue is thus the traditional data structure used in discrete event simulations, except for the additional consideration of breaking ties among events scheduled to occur at the same time. Any number of models or simulation events may be scheduled to execute at the same instantaneous timestep. However, the Model Coordinator may *not* then execute these events in an arbitrary order—there may be dependencies among them. For example, if the Household Mobility Model and the Household Location Choice Model are both scheduled to execute at time t, the Household Mobility Model must be run first, determining a set of households that decide to move, and then the Household Location Choice Model must be run to find available housing units for them. In other words, the choice to move from a current dwelling and the choice to look for a new dwelling are not independent; this dependency is reflected by the constraints on the order in which the models are run.

Since models are not restricted to running at regular intervals, in general it is not possible to determine execution orders until run-time. This introduces an enormous amount of complexity not found in most other urban modeling systems, which typically have a fixed ordering of execution. When more than one event is to occur at a given timestep, it is necessary to determine a total order of the events that preserves any order dependencies that may exist between them. Dependencies are of two types, data-level dependencies, and model-level dependencies between model execution events.

A data dependency exists between two models when one model writes to a field that another model reads from. In such cases, it is essential that the reading model reads the correct version of the data, and the writing model overwrites data only when it is safe to do so from another model's perspective. In the absence of other ordering dependencies, we assume that all reads to a field occur before any writes to it, and that writes can occur in any order. This reflects the typical access pattern of models, which generally read from many objects and write to a small number of fields of a small number of objects. (The fields written to generally have a very small overlap with reads from other models.)

A model-level dependency is an ordering dependency explicitly introduced by a model's author, in the form of a set of partial orderings between two or more models. This provides a mechanism to order models based on their semantics rather than their syntax (data reads/writes). For example, the Land Developer Model and the Land Price Model could be executed in either order, based on their inputs and outputs, but we schedule the latter to execute after the former so that adjustments in land price due to this year's demographic and economic changes do not affect development. This is based on the simplification of market dynamics that we adopt: that development decisions are made once per year, looking at the state of the region in the prior year to decide what should be build in the current year.

To determine a valid total order for a collection of events, a directed acyclic graph is constructed. Events are represented as nodes in the graph, and directed edges are created between events that access the same data fields of objects or that have model-level dependencies, with the direction of the edge indicating that the source node (event) must occur before the destination. When determining the possible dependencies between two model execution events, we must compute the transitive closure of all models and all fields that may potentially be read from or written to on the basis of the notification mechanisms. For example, model A may write to a field which is monitored by model B, which may in turn write to another field that triggers model C's notification mechanisms, and so forth. To ensure correctness, this full chain of potential reads and writes must be considered. Model-level dependencies override data-level dependencies in the case of conflicts. A topological sort is used to generate a valid total ordering of the events to be executed.

2.2.2 Development, Employment, and Policy Events

UrbanSim supports development events that create, modify, or delete buildings, create, move, or delete businesses or households, or change the urban growth boundary or re-zone land. Many of these events are generated by models. (For example, the developer model generates building development events.) However, events can also be read from an external file, allowing the modeler to introduce development projects and policy changes into the event pipeline that are exogenous to the models. For example, a scenario author may wish to simulate the effects of a major business leaving the region, a shopping center being constructed, or a modification to the urban growth boundary that occur at a particular point in time. This can be used for calibration purposes as well, by introducing major events from historical data. This capability was useful, for instance, in performing our historical validation of the system for Eugene-Springfield, Oregon (starting the model with 1980 data and comparing the simulated results for 1994 with what actually transpired). We used Development Events to model the phased closure of a large Weyerhauser lumber mill in the 1980's, and the opening of the Gateway regional shopping mall in 1990.

As a somewhat philosophical aside, the reader may wonder why these sorts of large events are exogenous and not produced by models themselves. The reason is that UrbanSim is intended to model the dynamics of a single urban region—but events in that region are influenced by the larger world. For example, the closing of a lumber mill might be due to declining timber stocks, changing world markets, or other external factors. Other examples of exogenous inputs are population control totals, based on predictions by demographers for the region, and overall economic predictions. This gives rise to another question. If we need to introduce a Business Event (such as a plant closure) for the system to give an accurate simulation of the historical development of a region, how can we have confidence in the system's predictions about a region in the year 2020? Might there not be some major event in 2015—for example, a global economic downturn—that will drastically influence the region? The answer is, of course, that UrbanSim provides no crystal ball regarding the global economy. Planners must use expert judgment in using the model, generally testing it under alternate scenarios and showing the results for all of them. For example, when evaluating the effects of a major transportation infrastructure project, it would be prudent to perform this evaluation based on several alternative scenarios for general economic conditions.

A related issue concerns policy events, such as moving an urban growth boundary or re-zoning land. These are also input as external events, and are not the output of a model. In contrast to development events representing global influences, these might be purely local policy changes. Our reason for representing policy events as exogenous inputs is philosophical rather than pragmatic: UrbanSim is intended as a tool to aid civic deliberation and debate, not as a tool to model the behavior of voters or governments. We want it to be used to say "if you adopt the following policy, here are the consequences," but not to say "UrbanSim predicts that in 5 years the city will adopt the following policy."

2.3 Object Store

The representations of agents in the world (such as households and businesses), and the objects they operate upon (such as buildings and land parcels), are held in the Object Store. The Object Store serves as an in-memory database that can be queried or updated, and that supports filtering on entity attributes.

The basic interface to the Object Store is through its postQuery and postUpdate methods. A model constructs a Query object by filling in the object type and set of fields to query for (e.g., the age and size category fields for households), and adds any Filter objects to the Query as desired (e.g., an EqualityFilter to return only households with a certain number of workers). The postQuery invocation returns a QueryResult object, which contains a copy of all relevant data from the Object Store, including internal object IDs for all values returned. Results are returned in parallel arrays (one array per field) to avoid Java object-level memory overhead. Updates work in a symmetric fashion, with a model constructing a Update object whose form is similar to QueryResult with the addition of the update type (create, modify, remove).

From a software engineering point of view, the Object Store also serves to encapsulate representation decisions about the entities in the simulation. From outside the Object Store, these act as traditional instances in an object-oriented language. However, they are represented more efficiently within the Object Store. Further, rather than defining these objects by writing a class definition, we use information in the Model Definition Files to give a description of just the portion of each object relevant to the corresponding model. These partial descriptions are then integrated by the system during the code generation phase to produce the eventual object definition.

2.3.1 Object Overhead Issues

Modeling a relatively small metropolitan region, such as Eugene-Springfield, requires some 350,000 objects to be represented in the Object Store. If these were represented as ordinary Java objects in our current Java implementation (Sun Java 2.0), there would be an additional overhead of approximately 20 bytes per object, for a total of seven megabytes. A larger region such as Salt Lake City requires some 1.5 million objects. Given the typical access patterns for the fields of objects (Section 2.3.3), performance is much improved if all the objects can be held in main memory, and so reducing space overhead is important.

Therefore, we represent objects efficiently within the Object Store, using parallel arrays holding the fields of each object type. For example, each Household object includes a integer field to hold its location, and a byte field to hold its income category. Rather than storing 300,000 explicit Household objects in the Object Store (with the attendant object overhead), we hold the information in a series of arrays, including an integer array Location with 300,000 elements, a 300,000 element byte array for income categories, and so forth.

Since these fields almost always hold primitive Java types such as **floats** or **ints** (rather than reference types), storing the fields directly in large arrays eliminates most of the space overhead. It also eliminates wasted space in each object due to word alignment padding, as would arise for byte fields. The encapsulation provided by the Object Store means that this non-standard representation is not visible outside it.

Other data structures have been implemented in a lightweight fashion, such as dynamic arrays, hash sets, and hash tables, to allow for storage of primitive types (**ints**, **floats**, etc.) in a fashion that eliminates the Java object-level overhead present from using Java's built-in data structures such as the ArrayList and HashMap.

```
class Zone
shared read int ZoneID
read float[] LandPricePerAcre
read float[] TotalAcresByALU
read float[] UsedAcresByALU
read float[] TotalValueByALU
read int[] TotalSqftByALU
read int[] UsedSqftByALU
```

Fig. 1. Definition of a Zone Object from the Developer Model.

```
class Zone
read int ZoneID
```

Fig. 2. Definition of a Zone Object from the default object definitions.

2.3.2 Construction of Object Class Definitions

Objects in the Object Store consist of the union of all fields defined by models for each object type and by the set of default object definitions (which are shared by most models). Queries can return copies of any of the fields of objects, and updates can modify fields or create or remove instances of objects. The Object Store's functionality has been tailored to the needs of UrbanSim-style models, including the ability to perform queries on spatially-correlated data (a task poorly performed by traditional databases).

The complete definition of each object type, and the Java code used to access and query it, is generated automatically from these partial object descriptions. For example, the Zone object type represents a traffic analysis zone. Partial definitions of Zone are given by both the default object definitions (Figure 2) and the Developer Model (Figure 1). These definitions are combined and used to generate the final version of the Zone object as defined in the DBArray.java file that contains the parallel-array storage of all object types (Figure 3), and query/update access methods are automatically generated as well (Figure 4). Other routines are automatically generated that enable objects to be saved or loaded from disk.

2.3.3 Swapping

The contents of the Object Store may be too large for the available main memory. To handle this, we provide a simple swapping mechanism that allows an array holding the contents of a field for an object type, e.g. the Location field from the Household type (Section 2.3.1), to be written out to disk if need be. This mechanism reflects the typical access patterns of models, which generally access every object of a given type, but only a small number of fields of each such object. (The more typical unit of swapping is the object—but given this access pattern, swapping on a per-object basis would be less desirable, since we would swap in entire objects, even most of the fields would not be immediately needed.) However, it is still preferable if possible to keep all data in main memory, since all of it is touched during each simulated year. Thus far, we have not used a commercial database as a back end, due

```
public class DBArray {
   ...
   // Fields for ZONE
   public static SimpleDynamicArrayI ZONE_ZoneID = null;
   public static SimpleDynamicArrayF ZONE_LandPricePerAcre = null;
   public static SimpleDynamicArrayF ZONE_UsedAcresByALU = null;
   public static SimpleDynamicArrayF ZONE_TotalAcresByALU = null;
   public static SimpleDynamicArrayF ZONE_TotalValueBYALU = null;
   public static SimpleDynamicArrayI ZONE_TotalSqftByALU = null;
   public static SimpleDynamicArrayI ZONE_UsedSqftByALU = null;
   ...
}
```

Fig. 3. Combined definition of Zone object (excerpt). Note the use of lightweight dynamic data structures, such as the SimpleDynamicArrayI array used to store integers while avoiding the Java object-level memory overhead from the ArrayList or Vector built-in data structures.

```
// Get the value of the corresponding field of an object, as an int
public static final int getFieldI(int objType, int objIdx, int fieldID) {
   ...
   switch ( objType ) {
      case DBObjTypes.ZONE:
         ...
         switch ( fieldID ) {
            case DBObjTypes.ZONE_ZONEID:
               return DBArray.ZONE_ZoneID.getI(objIdx);
            default:
               throw new RuntimeException("Get fieldID " + fieldID
                  + " not found for object type ZONE, field type int");
         }
      ...
   }
}
```

Fig. 4. Automatically generated Zone Object accessor methods (excerpt)

to a desire to not tie the Open Source UrbanSim code to a proprietary system. However, we plan to offer this as an option (but not a requirement) in a future version.

2.4 Translation and Aggregation Layer

The Translation and Aggregation Layer (T/AL) is responsible for converting between different levels of spatial and/or temporal aggregation from queries or updates and the objects in the Object Store. For example, models can query for zonal population totals. The Translation/Aggregation Layer computes and maintains these totals independent of the information in the Object Store, which consists of population information at the grid cell level.

At present, the T/AL is implemented using a set of methods in the Model Coordinator, rather than as a separate component, and serves only to cache query results for data aggregated at the zonal level. The two key methods that implement the T/AL, both in ModelCoordinator, are postQuery and postUpdate, to post a query or an update to the Object Store respectively. However, as the system evolves and we integrate models from increasingly diverse domains, we expect the T/AL to become increasingly important as we need to share data at widely different levels of spatial or temporal aggregation. At that time we may re-implement it as a separate component in its own right.

2.5 Data Import and Export

On the input side, UrbanSim requires a substantial amount of data to simulate an urban area, including census, employment, parcel, zoning, and transportation network data. We have written a suite of custom data preparation tools that can convert data to the formats we require, merge datasets, detect inconsistencies, and in some cases fill in missing data or correct errors using heuristics. This input data is generally made available to UrbanSim in the form of delimited ASCII text files.

On the output side, UrbanSim's output module is responsible for gathering, aggregating, and exporting data from the Object Store to a set of external files. These files are then passed on to other systems (including GIS software, statistical software, spreadsheets, and the external travel model) for subsequent analysis and graphical display. To simplify the software engineering aspects of data export, we have defined the output module as a model, analogous to the Demographic Transition Model, the Household Mobility Model, and so forth (Section 2.1). This allows the Export Model to be scheduled to run at defined times, querying the Object Store to take a snapshot of the state of the simulation. Conceptually, however, it is not a model like the Demographic Transition Model or others, since it does not represent an agent in the simulated world, and only performs reads of the Object Store, not writes.

Finally, in addition to input and output files, the system can write a snapshot of the current state of the Object Store to an external file, allowing the system to be restarted quickly from a given point in the simulation.

3 Experience and Lessons Learned

The previous version of UrbanSim (Waddell, 1998a) was a collection of tightly-integrated component models, including a developer, economic and demographic transition components, a land use component, and an external transportation model. The functionality of each model was intermingled with the functionality of the others, creating a large, complex system that did not lend itself to specialization, refinement, or enhancement. In creating a new framework for the UrbanSim model we sought to meet the following requirements:

- agent-level microsimulation of choice behavior
- a grid-based structure to represent spatial information, to facilitate detailed spatial queries and simulation
- easy replacement of one model by a new version, to support system evolution
- easy integration of new models
- support for different temporal and spatial scales
- support for visualization of the model output and its processes, for explanations and debugging

The new architecture has met these requirements. It has also proven relatively robust and stable, and has supported extensive model evolution (over nineteen versions of the Developer Model, for example), the introduction of several additional model components (splitting the location choice process from a unified market clearing process to separate residential and employment location choice models), the conversion from a business-centric to an employee-centric view of employment, and integration with an external, concurrently-running visualization environment (Pinnel et al., 2000).

Perhaps the most important software lesson learned from this work has been the value of moving as much of the complex functionality out of the individual models and into the supporting infrastructure as possible; most of the specific lessons discussed below are ways of achieving this goal.

3.1 Implicit Invocation

In the current UrbanSim architecture, models do not communicate directly with each other, but rather by registering interest in objects and fields held in the Object Store, and being notified when such an object or field has been changed by another model. The architecture thus uses a form of implicit invocation (Garland and Shaw, 1993; Sullivan and Notkin, 1992; Sullivan et al., 1996), in which components interact by generating and responding to events, rather than by explicitly invoking each other's methods. As in other systems, implicit invocation has proven to be a powerful technique for addressing component interaction complexity in UrbanSim. Two advantages of using implicit invocation have been:

- the ability to decouple models, since each model registers interest in objects and fields, makes changes to the Object Store, and responds to changes independently of the other models. This has made it significantly easier to experiment with new models and to evolve existing ones.
- ensuring a consistent interface for model interactions, since all interactions occur via the Object Store.

While our implicit invocation mechanism has worked well for the current style of models, we expect that it would break down if we moved to a much finer-grained simulation, for example, one in which households potentially moved, developers began construction of new buildings, and so forth, at any time, rather than on a yearly basis. The anticipated problems arise from the current mechanism being relatively coarse-grained: in the current architecture, a model cannot specify that it wishes to monitor a field in a particular object, only that it wishes to monitor a field in all objects of a given type. Any additional filtering or selection must occur within the callback code that is executed within the model as a result of the notification. This has been quite acceptable in the current system, but could introduce excessive numbers of unnecessary notifications in a more fine-grained approach, leading to unacceptably slow execution speeds.

A related difficulty is that the complexity of the callback code increases as more models use it to monitor and update related fields. For example, adding a new model that affects the existing functionality embedded in notification methods requires that the callback code in the new model take into account all of the existing functionality and not override or invalidate any of its actions.

3.2 Explicit versus Implicit Execution Ordering

We originally intended to handle most of the specification of model execution ordering using implicit data-level dependencies (Section 2.2.1). However, experience showed that the simple data-level ordering dependency rules (all reads to a field before any writes to it) failed to capture many important semantic constraints on model execution ordering. Thus in practice, the bulk of ordering dependencies are specified explicitly by model creators.

We also allow user-supplied model-level ordering dependencies to override data-level ones. An example where this is used is with the Export Model, which only reads from the Object Store and never writes to it. The implicit data-level dependencies would require that the exporter run before other models. However, we use an explicit ordering constraint to require that it be run *after* all the other models, so that it exports the information after the simulation has completed for the current simulated instant of time. However, if there are ordering dependency conflicts at the same level (i.e., at the model-level, or at the data-level in the absence of any overriding model-level dependencies), an error will be signaled.

The current design has worked well in practice, but is not entirely satisfying, and may begin to cause problems if the number of models increases dramatically. At that point we expect we will need to re-examine the issue, and perhaps find more sophisticated ways to determine implicit ordering constraints that reduce the need for explicit orderings.

3.3 Object Storage and Representation

Java's overhead for object representation (specifically, the class tag and the overhead of word alignment padding) has made the overhead of using standard Java objects in the Object Store prohibitive. Instead, we use a nonstandard object representation, namely parallel arrays holding the fields of each object type. However, this is not visible outside of the Object Store's encapsulation boundary.

This solution is not new to UrbanSim: it is akin to the use of marshalling in Smalltalk, CORBA, Java, etc. It is also more loosely related to the concept of flyweight objects (Calder and Linton, 1990; Gamma et al., 1995), although unlike flyweight objects, objects in the UrbanSim Object Store do not have extrinsic state that depends on their context.

The approach has had enormous benefits in the context of UrbanSim. Memory requirements have decreased by a factor of seven or more (as compared to the original object-based implementation), making the simulation of large areas such as Salt Lake City, Utah feasible on modest desktop systems. It has also facilitated the addition of disk-based swapping to the Object Store to further decrease memory overhead.

3.4 Automatic Generation of Code from Declarative Specifications

We have made considerable use in UrbanSim of the technique of generating Java code automatically from declarative specifications. One example is the generation of object class definitions for objects in the Object Store, along with query and update methods (Section 2.3.2).

The automatic generation of Java code that defines objects and their interface with the Object Store removes much programming burden from experimenters who introduce a new model or replace an existing one, as they do not have to write code that defines or manipulates object types. (This is particularly important since we use a non-standard representation of objects within the Object Store.) It also ensures that a consistent interface exists between the Object Store and every object that is contained within it.

We have also used automatic generation of Java code to define specialized, lightweight data structures optimized to store primitive Java types (**ints**, **floats**, etc.) with a minimum of memory overhead. These include lightweight dynamic arrays (SimpleDynamicArray), lightweight hash tables and sets (HashTableIntInt, HashSetInt, HashSetFloat, etc.), and some additional wrapper objects.

The issue addressed by these lightweight data structures is that in the standard Java library there is just one class definition for e.g. HashSet, whose element type is Object. Therefore, to store **ints** in a standard hash set, each **int** must be wrapped using the Integer class, leading to a considerable overhead in both space and time. Our automatically-generated HashSetInt class eliminates this overhead. If sufficiently powerful generic types were incorporated in future versions of Java, the need for these specialized lightweight data structures would be eliminated (or more precisely, they would be generated by the Java system itself rather than by us). Note that to be useful for our purposes, such a design must provide heterogeneous, not homogeneous, translation of generic types—homogeneous translation does not handle the space overhead problem. (In heterogenous translation, the compiler produces different versions of the code for each instantiated type; in homogeneous translation, just one version. For Java, homogeneous translation requires wrapping each primitive type, which adds a tremendous space penalty. Both varieties of translation are available in the Pizza extension to Java (Odersky and Wadler, 1997); GJ (Bracha et al., 1998) and NextGen (Cartwright and Steele Jr., 1998) provide only homogeneous translation.)

3.5 Choice of Programming Language

UrbanSim is implemented in the Java programming language. Java has provided a solid environment for implementing a system of this kind, particular strengths being automatic storage management, static typing, a rich class library, and portability. While its execution speed is not comparable to C++, current compiler technology (e.g., Just-In-Time compilation) has provided reasonable performance. Perhaps the biggest problem for us has been the overhead of object representation, which has required unorthodox object representations within the Object Store (Section 2.3).

Java provides many advantages over other programming languages that were considered. These advantages include:

- Object-oriented language, where we can use inheritance and subclassing to increase reusability of language-level objects and code we write;
- Automatic storage management, which helps improve software reliability by eliminating memory allocation and storage errors that can be difficult to find and fix in other environments;
- A rich collection of data structures and libraries, which improves productivity when writing code;
- Portability, so that we can run the UrbanSim model on a variety of platforms without needing to modify the source code;
- Performance, which is reasonably good due to the use of Just-In-Time (JIT) compilers, and is substantially better than interpreted languages.

We considered using one of the dedicated modeling languages or environments which are available, such as Stella, Swarm (Luna and Stefannson, 2000; Swarm Development Group, 2000), StarLogo (StarLogo Group, 2001), or the MIMOSE framework (Möhring, 1996). However, none of these met our requirements for portability, generality, and efficiency. We thus chose to implement a framework from scratch, enabling us to customize and optimize it for the specific domain and types of models we work with.

3.6 An Open Source Software Approach

The UrbanSim software is licensed under the GNU General Public License (GPL) from the Free Software Foundation. This license is well-known among software developers, and is used for such systems as the linux operating system and the emacs text editor. It is not so commonly used for software for urban modeling, but has some significant benefits. The license allows anyone who has the software to further distribute it, to change it, or to use parts of it in new programs. Further, it requires that any distribution of the system or of derivative works continue to be licensed under the GPL. The distribution must include the source code as well as the compiled object code, or make the source code easily available. (Please see the GPL itself (Free Software Foundation, 1991) for the precise terms of the license.)

The benefits of licensing UrbanSim under the GNU General Public License are that other researchers and practitioners can freely apply the software, as well as build on it and distribute those results as well. In contrast, a proprietary license would typically require the payment of fees, and might well not include the source code. Using the GPL license also has quite different implications from placing the source code in the public domain: if this were done, the version in the public domain would be freely available, but another could produce a derivative work (perhaps involving changes to a relatively small percentage of the system), and then place this derivative work under a restrictive, proprietary license.

The original implementation work was done by a single programmer (the first author). Subsequently, however, portions of the infrastructure and many of the models have been modified or created by others. We use the Concurrent Versions System (CVS) (Cederqvist, 2000) as a means of coordinating and integrating programming work performed by the different programmers that work on this project. CVS maintains a central repository with the definitive copy of the source code. A programmer "checks out" a copy of the system, which can be freely modified on that programmer's personal disk directory. When the new version is ready, the programmer can then check it in to the repository. CVS keeps track of the successive versions of the system, and watches for conflicting check-ins by different programmers, notifying them so that the differences can be resolved.

CVS is often used for Open Source development projects, since it is admirably suited to coordinating the work of programmers at geographically distributed locations. Another feature of CVS is that the developers can allow anyone read-only access to the repository, so that others can track the latest versions of the software. We have not yet used these features, but plan to in the near future as a way of coordinating work with a Metropolitan Planning Organization that will be applying UrbanSim to that region.

4 Related Work

As described in Section 3.1, the UrbanSim architecture uses a form of implicit invocation, in which models communicate with each other indirectly, by registering interest in objects and fields held in the Object Store, and by being notified when such an object or field has been changed by another model. Additional advantages and other applications of implicit invocation are described in references Garland and Shaw (1993); Sullivan and Notkin (1992); Sullivan et al. (1996). Implicit invocation is essentially an event mechanism; related concepts include active variables in LOOPS (Stefik et al., 1986), active databases such as AP5 (Cohen, 1989), and the Smalltalk-80 Model-View-Controller (Krasner and Pope, 1988) and Field integration mechanisms (Reiss, 1994). A discipline of defining and using event-based programming mechanisms is evolving (Barrett et al., 1996; Carzaniga et al., 1998; Garlan and Notkin, 1991).

There is a huge body of work on urban transportation modeling, land use modeling, and integrated land use/transportation modeling. Reviews and assessments of existing systems are given in references Miller et al. (1998); Parsons Brinckerhoff Quade and Douglas (1998); Southworth (1995), among others. Considerable progress has recently been made in land use modeling in both experimental and deployed systems. However, except for UrbanSim, all the operational models in use by planning agencies rely on a cross-sectional, aggregate, equilibrium approach. Such models include DRAM/EMPAL (Putman, 1983), TRANUS (de la Barra, 1995), MEPLAN (Echenique et al., 1990), METROSIM (Anas, 1994), and 5-LUT (Martinez, 1992). The cross-sectional, equilibrium framework implies that there are no relevant temporal dynamics to the processes of urban change; rather, one can model urban development as a static process that represents an economic or a transportation optimization problem. In other words, these models could be run for the year 2050 without needing to model the dynamics of evolution between the current time and the year 2050. Clearly, this is a severe simplification, and makes problematic the potential integration of these models with models of dynamic environmental processes, or even of the dynamic evolution of human behavior with respect to the built environment.

Another substantial body of related work concerns Integrated Assessment Models (IAMs), which model the interactions between human and ecological systems in an integrated way. A major motivation for models of this kind is the assessment of global environmental change (Alberti, 1999; Dowlatabadi, 1995; Parson and Fisher-Vanden, 1995; Rotmans et al., 1995; Weyant et al., 1996). While the first generation of operational IAMs has emerged in the mid-eighties, their roots can be traced back to earlier modeling work in the late sixties and early seventies (Forrester, 1971; Isard, 1969; Meadows et al., 1982; Odum, 1983). Not surprisingly, all of these global-scale models are quite aggregate, predicting environmental disturbances from broad measures of economic growth and urbanization.

In addition to global models, spatially-explicit regional integrated models are now emerging, such as the Patuxent Landscape Model (Voinov et al., 1999). The Patuxent Landscape Model contains an economic land use conversion model that uses a statistical process to determine probabilities that grid cells will be allocated to forest, agricultural, or urban usage. The resulting conversion probabilities are used to predict land use patterns which determine the land cover values used as an input to the PLM's hydrology component. Communication between the land conversion and hydrology models is implicit through changed data values in grid cells. Several of the factors used in its land use conversion component are similar to ones used in UrbanSim (e.g., access to infrastructure, historical tax assessor data), but UrbanSim explicitly models agents and their actions rather than using statistical or finite-element processes.

Finally, another area of related work concerns agent-based modeling, artificial life, and cellular automata. In agent-based modeling in its pure form, individual agents and their actions are simulated, with each agent having local knowledge; global behavior then emerges from these agent-level interactions. Agent-based modeling has been used for a wide range of applications, including economic, sociological, biological, and physical simulations. Two that are most closely related to UrbanSim are Sugarscape (Epstein and Axtell, 1996; Brookings Institution, 2000), a simulation of a small, artificial society, and Aspen, a microanalytic simulation of the entire U.S. economy (Pryor et al., 1996; Sandia National Laboratories, 2000).

A number of the UrbanSim models, such as the mobility, location choice, and development models, are agent-based. However, we have not adopted this approach throughout the system, and have used more global models when appropriate—our principal interest is in the realistic simulation of urban development to inform public policy, by whatever techniques give the best results, rather than in the investigation of whether a particular modeling technique can encompass all aspects of the simulation. For example, the demographic transition model uses a single module to compute the births and deaths in the simulated population of households using externally imposed population control totals, along with demographic data regarding distribution by income groups, age, and so forth. In a pure agent-based approach, individual agents would decide whether to form households, have children, move out of the region, and so forth. While such an approach can yield patterns of global behavior that exhibit many of the characteristics of the real world, in our application we have the much more stringent requirement of matching the population characteristics of a region, including reproducing historical data, and the pure agent-based approach has not appeared practical in such cases.

Cellular automata have been used for simulating urban development (Batty, 1998, 1999; Clarke et al., 1997), as well as for other applications such as simulating change in land cover, freeway traffic, or the spread of wildfires. In its classic form, a cellular automaton consists of a regular array of cells, each of which has a finite number of states. Each state change must be local, depending only on the states of neighboring cells. Urban processes, such as sprawl or urban decay, can emerge from simple local rules. However, these restrictions do not always mesh well with our goal of supporting deliberation about public policy. For example, rather than viewing the conversion of rural areas to urban ones as an analog of a biological process in which the suburb grows and occupies increasingly wider areas, in UrbanSim we view this process as the result of interactions among the Land Developer Model (which simulates developers actively seeking out development opportunities throughout the region in response to market conditions, zoning regulations, taxes and incentives, and the like), the location choice models (which simulate residents or businesses seeking housing and commercial space), the Land Price Model, and so forth. More recently, researchers have experimented with extensions of the cellular automata formalism that incorporate extensions such as more agent-like behavior or non-local search (Batty and Jiang, 1999; O'Sullivan and Torrens, 2000).

5 Conclusion

UrbanSim is both a vehicle for research on modeling urban systems and a practical system that has been used in planning work in several U.S. cities. The software uses a modular architecture that allows models to be written as independently as possible, and that provides for a clean separation between the models and the data on which they operate. This has supported extensive experimentation with alternate modeling techniques, and also supported implementing a system that is much more disaggregate and responsive to policy considerations than others of its kind. The primary software lesson learned from this work has been the value of moving as much of the complex functionality out of the individual models and into the supporting infrastructure as possible; specific techniques for achieving this include implicit invocation, efficient object representation in an encapsulated object store, and extensive use of automatic generation of code from declarative specifications.

Acknowledgments

Thanks to Nathan Freier for comments on this paper. Many people have contributed to UrbanSim. We would particularly like to thank Michael Becke and Nathan Freier for work on the software, as well as A.J. Brush, David Chang, Matthew Dockrey, Leo Lai, Dorian Miller, and Denise Pinnel. Thanks to Gudmundur Ulfarsson, Kevin Krizek, John Carruthers and Robert Willis for work on data analysis and model calibration. This research has been funded in part by National Science Foundation Grant Nos. CMS-9818378 and IIS-9975990, and in part by the University of Washington PRISM project.

References

Alberti, M., 1999. Modeling the urban ecosystem: A conceptual framework. Environment and Planning B 26, 605–630.

Anas, A., 1994. METROSIM: A Unified Economic Model of Transportation and Land-Use. Alex Anas & Associates, Williamsville, NY.

Barrett, D., Clarke, L., Tarr, P., Wise, A., Oct. 1996. A framework for event-based software integration. ACM Transactions on Software Engineering and Methodology 5 (4), 378–421.

Batty, M., 1998. Urban evolution on the desktop: Simulation with the use of extended cellular automata. Environment and Planning A 30, 1943–1967.

Batty, M., 1999. Modelling urban dynamics through GIS-based cellular automata. Computers Environment and Urban Systems 23, 205–233.

Batty, M., Jiang, B., Apr. 1999. Multi-agent simulation: New approaches to exploring space-time dynamics within GIS. Tech. Rep. 10, Centre for Advanced Spatial Analysis, University College London.

Beckman, R. J., Baggerly, K. A., et al., 1995. Creating synthetic baseline populations. In: Transportation Research Board Annual Meeting. Washington, D.C.

Bracha, G., Odersky, M., Stoutamire, D., Wadler, P., 1998. Making the future safe for the past: Adding genericity to the Java programming language. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 183–200.

Brookings Institution, 2000. Sugarscape home page. Web page: `http://www.brook.edu/sugarscape`.

Calder, P., Linton, M., 1990. Glyphs: Flyweight objects for user interfaces. In: Proceedings of the ACM Symposium on User Interface Software Technology. pp. 92–101.

Cartwright, R., Steele Jr., G. L., 1998. Compatible genericity with run-time types for the Java programming language. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 183–200.

Carzaniga, A., Di Nitto, E., Rosenblum, D., Wolf, A., Nov. 1998. Issues in supporting event-based architectural styles. In: Proceedings of the Third International Software Architecture Workshop.

Cederqvist, P., 2000. Version Management with CVS. iUniverse.com.

Clarke, K. C., Hoppen, S., Gaydos, L., 1997. A self-modifying cellular automaton model of historical urbanization in the San Francisco Bay area. Environment and Planning B: Planning & Design 24, 247–261.

Cohen, D., 1989. Compiling complex transition database triggers. In: Proceedings of ACM SIGMOD. Portland OR, pp. 225–234.

de la Barra, T., 1995. Integrated Land Use and Transportation Modeling: Decision Chains and Hierarchies. Cambridge University Press.

Dowlatabadi, H., 1995. Integrated assessment models of climate change: An incomplete overview. Energy Policy 23 (4/5), 289–296.

Echenique, M., Flowerdew, A., Hunt, J., Mayo, T., Skidmore, I., Simmonds, D., 1990. The MEPLAN models of Bilbao, Leeds and Dortmund. Transport Reviews 10 (4), 309–322.

Epstein, J. M., Axtell, R. L., 1996. Growing Artificial Societies: Social Science from the Bottom Up. Brookings Institution Press & MIT Press, Washington, D.C.

Forrester, J., 1971. World Dynamics. MIT Press, Cambridge, MA.

Free Software Foundation, 1991. GNU general public license, version 2. Web page: `http://www.gnu.org/copyleft/gpl.html`.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Garlan, D., Notkin, D., 1991. Formalizing design spaces: Implicit invocation mechanisms. In: Proceedings of VDM Europe. pp. 31–44.

Garland, D., Shaw, M., 1993. An introduction to software architecture. In: Advances in Software and Knowledge Engineering. Vol. 2. World Scientific Publishing Company.

Garret, M., Wachs, M. (Eds.), 1996. Transportation Planning on Trial: The Clean Air Act and Travel Forecasting. Sage Publications, Thousand Oaks, CA.

Isard, W., 1969. Some notes on the linkages of ecological and economic systems. Papers of the Regional Science Association 22, 85–96.

Krasner, G., Pope, S., August/September 1988. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. Journal of Object Oriented Programming 1 (3).

Lee, D. B., 1973. Requiem for large-scale models. Journal of the American Institute of Planners 39, 163–178.

Lee, D. B., 1994. Retrospective on large-scale urban models. Journal of the American Planning Association 60 (1), 35–40.

Luna, F., Stefannson, B. (Eds.), 2000. Economic Simulations in Swarm: Agent-Based Modelling and Object-Oriented Programming. Kluwer Academic Publishers.

Martinez, F., 1992. The bid-choice land use model: an integrated economic framework. Environment and Planning A 24, 871–885.

Meadows, D., Richardson, J., Bruckmann, G., 1982. Groping in the Dark: The First Decade of Global Modeling. John Wiley & Sons, New York.

Miller, E., Kriger, D., Hunt, J., 1998. Integrated urban models for simulation of transit and land use policies. Final Report, Project H-12. TCRP Web Document 9, Transit Cooperative Highway Research Project, National Academy of Sciences: Washington, DC. `http://books.nap.edu/books/tcr009/html`.

Möhring, M., 1996. Social science multilevel simulation with MIMOSE. In: Troizsch, K. G., et al. (Eds.), Social Science Microsimulation. Springer, Berlin, pp. 282–306.

Odersky, M., Wadler, P., 1997. Pizza into Java: Translating theory into practice. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 146–159.

Odum, H., 1983. Systems Ecology: An Introduction. John Wiley & Sons, New York.

O'Sullivan, D., Torrens, P., 2000. Cellular models of urban systems. In: Theoretical and Practical Issues on Cellular Automata: Proceedings of the Fourth International Conference on Cellular Automata for Research and Industry (ACRI 2000). Springer-Verlag.

Parson, E. A., Fisher-Vanden, K., 1995. Searching for integrated assessment: A preliminary investigation of methods and projects in the integrated assessment of global climatic change. CIESEN-Harvard Commission on Global Environmental Change Information Policy.

Parsons Brinckerhoff Quade and Douglas, 1998. Land use impacts of transportation: A guidebook. Transportation Research Board, National Research Council.

Pinnel, L. D., Dockrey, M., Brush, A. B., Borning, A., May 2000. Design of visualizations for urban modelling. In: Proceedings of VISSYM '00 — Joint Eurographics - IEEE TCVG Symposium on Visualization. Amsterdam.

Pryor, R. J., Basu, N., Quint, T., Feb. 1996. Development of Aspen: A microanalytic simulation model of the U.S. economy. Tech. Rep. SAND96-0434, Sandia National Laboratories.

Putman, S., 1983. Integrated Urban Models: Policy Analysis of Transportation and Land Use. Pion, London.

Reiss, S., 1994. FIELD: A Friendly Integrated Environment for Learning and Development. Kluwer.

Rotmans, J., Dowlatabadi, H., Filar, J., Parson, E., 1995. Integrated assessment of climate change: Evaluation of methods and strategies. In: Institute, B. (Ed.), Human Choices and Climate Change: A State of the Art Report. Battelle Pacific Northwest Laboratory, Washington D.C.

Sandia National Laboratories, 2000. Aspen: A smart, agent-based economics simulation model. Web page: `http://www-aspen.cs.sandia.gov`.

Southworth, F., 1995. A technical review of urban land use-transportation models as tools for evaluating vehicle reduction strategies. U.S. Department of Energy.

StarLogo Group, 2001. StarLogo on the web. Web page: `http://el.www.media.mit.edu/groups/el/Projects/starlogo/`.

Stefik, M., Bobrow, D., Kahn, K., Jan. 1986. Integrating access-oriented programming into a multiparadigm environment. Software .

Sullivan, K., Kalet, I., Notkin, D., Aug. 1996. Mediators in a radiation treatment planning environment. IEEE Transactions on Software Engineering 22 (8).

Sullivan, K., Notkin, D., 1992. Reconciling environment integration and software evolution. ACM Transactions on Software Engineering and Methodology 1 (3), 229–268.

Swarm Development Group, 2000. Web page: `http://www.swarm.org`.

UrbanSim Group, Aug. 2000. UrbanSim reference guide, version 0.9. Available from `http://www.urbansim.org`.

Voinov, A., Costanza, R., Wainger, L., Boumans, R., Villa, F., Maxwell, T., Voinov, H., 1999. Patuxent landscape model: Integrated ecological economic modeling of a watershed. Environmental Modelling and Software Journal 14 (5), 473–491.

Waddell, P., May 1998a. The Oregon prototype metropolitan land use model. In: 1998 ASCE Conference on Transportation, Land Use and Air Quality: Making the Connection. Portland, Oregon.

Waddell, P., 1998b. Simulating the effects of metropolitan growth management strategies. In: 1998 Conference of the Association of Public Policy Analysis and Management. Available from `http://www.urbansim.org`.

Waddell, P., Jul. 2000. Historical validation in Eugene-Springfield. Presentation at Second Oregon Symposium on Integrating Land Use and Transport Models, available from `http://www.urbansim.org`.

Weyant, J., et al., 1996. Integrated assessment of climate change: A overview and comparison of modeling approaches and results. In: Bruce, J., Lee, H., Haites, E. (Eds.), Climate Change 1995: Economic and Social Dimensions of Climate Change. Cambridge University Press, Ch. 10, pp. 367–396.

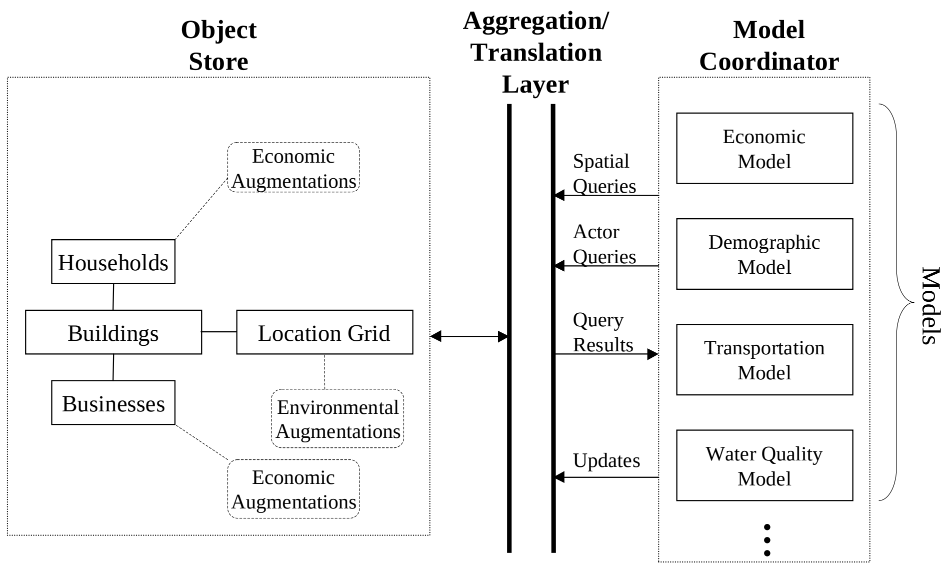

Figure 5: UrbanSim architecture.